by
Ray Duncan

Power Programming

Strategies for Optimizing Assembly Language Programs

■ To help your assembly language programs deliver maximum performance, sometimes you'll want to optimize for size, sometimes for speed, and sometimes you're better off leaving well enough alone.

In spite of the increasing dominance of high-level languages and integrated development environments, the optimization of assembly language code is still a fashionable subject for on-line debates. Thus the Programming Forum on PC MagNet has been the venue for many "coding challenges": one member describes a small but interesting programming problem, then sits back to see who can solve it in the fewest number of bytes. Similarly, the assembler/cpu8088 conference on BIX has been the site of many grave deliberations on obscure assembly language optimization issues.

Grass-roots interest notwithstanding, the literature on Intel 80*x*86 assembly language optimization is surprisingly sparse. While preparing a talk on this subject for the Software Development Conference a couple of years ago, I searched the back issues of the major programming magazines and found only a handful of articles. The literature on code optimization for high-level language compilers, on the other hand, is quite extensive, and many of the concepts developed in this literature are useful in assembly language coding as well. I've included a bibliography for those of you who wish to explore this subject further (see the sidebar "Further Reading"). In the meantime, let's survey some of the more common optimization techniques and then turn to the larger question of when and where to optimize.

OPTIMIZING FOR SPEED

When you decide that a program isn't running fast enough, the first thing to do is to make sure that you are using the best algorithms and data representations for the job. Switching from an inappropriate or primitive algorithm to one that's more fitting or clever may well speed up your program by an order of magnitude or more, so taking the time to peruse Knuth or Sedgewick for an algorithm you wouldn't be likely to think of on your own is time well invested. Similarly, changing from an "obvious" but simple data structure (such as a linked list) to a more sophisticated structure (such as a B-tree) may yield results far out of proportion to the programming effort involved.

Once you're satisfied that you're using the best algorithms and data structures, the next thing to look at is the program's use of mass storage. Even the fastest disk devices used on personal computers are abominably slow compared with the computing horsepower of an 80386 or 80486, so you'll want to minimize or eliminate disk I/O whenever possible. Become familiar with all the types of memory available to a DOS program—extended memory, expanded memory, upper memory blocks, and the high memory area—and fully exploit this memory to keep your program's data in RAM where you can get at it quickly. You may even want to look into data compression techniques, because it will almost always be faster to squeeze data and unsqueeze it again than to read it two or more times from disk.

Another important area to address is reducing your program's calculation burden. The compiler writers have fancy terms—*common subexpression elimination, constant folding, constant propagation*—for variations on the basic rule that your program should never calculate the same value twice at runtime; instead, it should calculate the value once, and save it for reuse. An even better solution is to move calculations from runtime to assembly time whenever the limited mathematical abilities of MASM (or TASM or OPTASM) allow. Along the same lines, you may be able to enhance a program's performance drastically by converting calculations to table lookups, and making use of tables that are generated and stored on-disk ahead of time by a separate program.

Once you've done whatever can be done at these more abstract levels, you're pretty much reduced to classic code hacking and cycle shaving to improve your program's performance—particularly if your program is heavily display-oriented and performs its own memory-mapped video output. Some of the more common optimizations in this category are:

- Substitution of fast special-case sequences of instructions for more generalized, complex instructions, such as replacing a hardware multiply by a power of two with a shift or series of shift instructions (strength reduction).
- Reducing the overhead of jumps and calls by converting subroutines to macros and assembling them in-line; changing the sense of conditional jumps so that the jumps are taken on the less common condition; moving common cases to the

FURTHER READING

by Ray Duncan

The following are selected readings on code optimization for assembly language programmers.

BOOKS:

- *Compilers: Principles, Techniques, and Tools*, chapter 10, by Alfred Aho, Ravi Sethi, and Jeffrey Ullman, Addison-Wesley Publishing Co., Reading, Mass., 1986.
- *Crafting a Compiler*, chapters 15 and 16, by Charles Fischer and Richard LeBlanc, Benjamin/Cummings Publishing Co. Inc., Menlo Park, Calif., 1988.
- *Programming Pearls*, chapters 3, 5, 8, and 9, by Jon Bentley, Addison-Wesley Publishing Co., Reading, Mass., 1986.
- *Writing Efficient Programs*, by Jon Bentley, Prentice Hall Publishing Co., Englewood Cliffs, N.J., 1982.
- *The Zen of Assembly Language*, by Michael Abrash, Scott Foresman & Co., Glenview, Ill., 1990.

ARTICLES:

- "Assembly Language Tricks of the Trade," by Tim Paterson, *Dr. Dobb's Journal*, March 1990, page 30.
- "Loop Unrolling," by Mark Barrenechea, *Programmer's Journal*, July/August 1991, page 66.
- "More Optimizing for Speed," by Michael Abrash, *Programmer's Journal*, July/August 1986, page 36.
- "Optimania," by Mark Barrenechea, *Programmer's Journal*, January/February 1991, page 72.
- "Optimization Strategies," by Gary Sarff, *Computer Language*, December 1985, page 27.
- "Optimizing C with Compiler-Generated Assembly," by John Ross, *Computer Language*, November 1988, page 59.
- "Optimizing Compilers," by Mark Roberts, *BYTE*, October 1987, page 165.
- "Optimizing Compilers for C," by Richard Relph, *Dr. Dobb's Journal*, August 1987, page 42.
- "Optimizing for Speed," by Michael Hoyt, *Programmer's Journal*, March/April 1986, page 32.
- "Optimizing Integer Multiplications by Constant Multipliers," by Robert Grappel, *Dr. Dobb's Journal*, March 1987, page 34.
- "Peak Performance," by Mark Barrenechea, *Programmer's Journal*, November/December 1990, page 64.
- "Pushing the Envelope," column by Michael Abrash in *PC Techniques*.
- "Roll Your Own Languages with Mini-interpreters," by Michael Abrash and Dan Illowsky, *Dr. Dobb's Journal*, September 1989, page 52.
- "Secrets of Optimization," by Walter Bright, *Micro Cornucopia*, January/February 1989, page 26.
- "8088 Assembly-Language Programming Techniques," by Tom Disque, *Dr. Dobb's Journal*, July 1987, page 24.
- "80x86 Optimization," by Michael Abrash, *Dr. Dobb's Journal*, March 1991, page 16.

front of multiway jumps; converting calls followed immediately by returns into jumps ("tail merging" and "tail recursion optimization"), and so on.

- Various loop optimizations including moving the calculations of invariant values outside of loops, simplifying calculations inside of loops, unrolling loops, and consolidating separate loops that are executed the same number of times into a single loop ("loop jamming").
- Taking maximum advantage of the available registers by keeping working values in registers whenever possible to minimize memory accesses, packing multiple values or flags into registers, and eliminating unnecessary stack pushes and pops (especially at subroutine entries and exits).
- Exploiting CPU-specific instructions such as the push immediate value instruction that is available on 80286 or later CPUs, or the doubleword string instructions, 32-bit by 32-bit multiply, 64-bit by 32-bit divide, and multiply by immediate value instructions that are available on 80386 and 80486 CPUs. Of course, your program must first determine which CPU it is running on! The routine shown in Figure 1 can help in this regard: It returns a code indicating CPU type.

On 80286 and later CPUs, you may also be able to improve execution speed by a few percent by aligning data items and the targets of jump instructions on appropriate boundaries. The 8086 and 80286 are 16-bit CPUs and like to see things aligned on 16-bit (word) boundaries; the 80386 prefers 32-bit (doubleword) alignment; and the 80486 performs best with paragraph (16-byte) alignment because of the design of its internal cache.

If all else fails and you're writing a specialized application rather than a mass-market software package, you can usually trump your performance problems with hardware. At today's mail-order prices, the cost of replacing the machine your application runs on with a new, much faster machine may be considerably less than the cost of the time required for you to painstakingly redesign, optimize, and re-debug the application. Unfortunately, the undiscriminating embrace of this strategy by vendors such as Microsoft and Lotus has brought us bloated monstrosities such as *Windows* 3.0, *Programmer's Workbench*, and *Lotus 1-2-3/G*, but that's a subject for another day.

OPTIMIZING FOR SIZE

If your program is too big to run (rather than too slow), you need to fall back on optimization strategies that are in most ways the exact inverse of the tricks you use for speed. First examine your program carefully to determine whether the primary problem is code size or data size.

If your data is too big, sophisticated data structures may help, but the elimination of fast but sparse arrays or tables in favor of more compact structures such as linked lists and the packing of data using bit fields will probably yield more short-term benefits. Brute force compression and decompression of tables or structures as they are needed is not usually very useful, because it's often necessary to

CPUTYPE.ASM — COMPLETE LISTING

```
        title   CPUTYPE -- check CPU Type
        page    55,132
        .486

; CPUTYPE.ASM - Returns Intel CPU type.  Adapted from
;               source code distributed to ISVs by Intel Corp.
;
; Call with:    N/A
;
; Returns:      AX = CPU type
;                    0086H = 8086 or 8088
;                    0286H = 80286
;                    0386H = 80386SX or 80386DX
;                    0486H = 80486SX or 80486DX
;
; Destroys:     upper 16-bits of EAX and ECX on 386/486

_TEXT   segment word use16 public 'CODE'

        assume  cs:_TEXT

        public  cputype
cputype proc    near

        pushf                           ; save copy of flags and
        push    bx                      ; other affected registers
        push    cx
        pushf                           ; now try to clear bits 12-15
        pop     ax                      ; of CPU flags
        and     ax,0fffh
        push    ax                      ; set modified CPU flags
        popf
        pushf
        pop     ax                      ; get flags again
        and     ax,0f000h               ; if bits 12-15 are still
        cmp     ax,0f000h               ; set, this is 8086/88
        jne     cpu1                    ; jump, not 8086/88
        mov     ax,0086h                ; set AX = 86/88 CPU type
        jmp     cpux                    ; and exit

cpu1:   or      ax,0f000h               ; must be 286 or later,
        push    ax                      ; now try to set bits 12-15
        popf                            ; of CPU flags
        pushf
        pop     ax                      ; if bits 12-15 can't be
        and     ax,0f000h               ; set, this is a 286
        jnz     cpu2                    ; jump, not 80286
        mov     ax,286h                 ; set AX = 286 CPU type
        jmp     cpux                    ; and exit

cpu2:   mov     bx,sp                   ; 386 or later, save SP
        and     sp,not 3                ; avoid stack alignment fault
        pushfd                          ; get value of EFLAGS
        pop     eax
        mov     ecx,eax                 ; save copy of EFLAGS
        xor     eax,40000h              ; flip AC bit in EFLAGS
        push    eax                     ; try and force EFLAGS
        popfd
        pushfd                          ; get back EFLAGS value
        pop     eax
        mov     sp,bx                   ; restore old stack pointer
        xor     eax,ecx                 ; can AC bit be changed?
        jnz     cpu3                    ; no, jump, not a 386
        mov     ax,0386h                ; set AX = 386 CPU type
        jmp     cpux                    ; and exit

cpu3:   mov     ax,0486h                ; set AX = 486 CPU type

cpux:   pop     cx                      ; restore registers
        pop     bx
        popf                            ; restore original flags
        ret                             ; return with AX = cpu type

cputype endp

_TEXT   ends

        end
```

Figure 1: When a program is running on an 80286, 80386, or 80486 CPU, there are many CPU-specific instructions that can be exploited to improve performance. First, however, the program must be able to determine the CPU type so it can adapt itself accordingly. This routine, adapted from code distributed by Intel, returns a code indicating whether the host CPU is an 8088, 8086, 80286, 80386, or 80486.

decompress all the data just to get at one em, and the compression/decompression routines themselves occupy considerable memory.

Large lookup tables and arrays may be built in a disk file and read into memory in small chunks when needed, but this can have a crushing effect on throughput if the data is randomly accessed. Lookup tables can often be completely eliminated in favor of recomputing values as they're needed. You must also search for and remove constants and variables that are never actually used by the program because they were superseded by other constants and variables during development and debugging. In any event, I'll stress again how important it is for you to learn how to access all the types of memory that may be available on a PC, and make your program flexible enough to use any and all of them.

Optimizing a program's code for size is quite a different ballgame than optimizing it for speed. First you must inspect every line of source code and eliminate all statements or procedures that are never executed or can't be reached from any point in the main line of execution (dead code). If you've got a large application that has gone through a prolonged development cycle involving several programmers, you may be quite surprised at the number of lines you can excise. Second, analyze the program again and consolidate all identical or functionally similar code sequences into subroutines that can be called from any point in the program. The more general you can make the subroutines, the more likely it is that their code can be reused. When carried to an extreme, this approach leads to a highly modular and compact program that consists mostly of calls.

Optimizing a program's code for size is quite a different ballgame than optimizing it for speed.

At this point, if there still isn't enough memory to go around, your efforts can diverge in several different directions. You can decompose your program into relatively independent modules that can be read into memory separately as overlays. You can encode your program's functionality in tables that "drive" its execution. You can resort to threaded-code or pseudo-code techniques that represent your program's logic in considerably less memory at some cost in performance. Or you can really pull out all the technological stops and implement a custom interpreter or compiler for a "little language" that in turn serves as the virtual machine for your application. QuickBASIC, *Excel*, and *BRIEF* are everyday examples of the threaded-code, pseudo-code, and little-language strategies, respectively.

Finally, if you're writing a special-purpose application, you may be able (again) to simply trump your memory problems with hardware or software. Among the many possibilities are: upgrading to DOS 5.0 or OS/2 2.0 to make more memory available below 640K, plugging in yet another EMS board, switching to a 80386 or 80486 machine that will support more extended memory than your 8086 or 80286 machine, and/or porting your program to run on a DOS extender.

WHEN TO OPTIMIZE, WHEN TO PUNT

Code optimization in any language always requires tradeoffs, such as:

- trading speed for memory;
- trading code maintainability and readability for code performance;
- trading development time for execution time.

These tradeoffs seem straightforward, but more often they aren't as simple as they first appear. The following two instructions, either of which can be used to transfer a value from register DX into AX (with different side effects, of course) provide a classic example:

```
XCHG    AX,DX

MOV     AX,DX
```

On an Intel 8088, the MOV instruction is 2 bytes and requires 2 CPU clock cycles, while the XCHG instruction is 1 byte but requires 3 CPU clocks. So far, a clear-cut tradeoff of speed for memory. But the actual performance of the instructions is highly dependent on context, the size of the CPU's prefetch queue, the size and characteristics of system caches, and so on, while even the number of clock cycles required to execute the instructions varies from one model of Intel CPU to another. As it turns out, it's nearly impossible to predict the performance of a nontrivial sequence of instructions on an Intel processor, especially on the later CPUs such as the 80386 and 80486, which make more extensive use of pipelining; you simply have to benchmark the various possibilities in vivo and see how they perform.

Similarly, the tradeoffs of development time for execution time and code maintainability for performance are seldom as well ordered as we'd like, and the long-term impact of wrong decisions can be enormous. By far the most important thing to understand about optimization is when to do it and when to leave well enough alone; as Donald Knuth said, "Premature optimization is the root of much programming evil." Before you can even think about tuning up your program, make sure it is logically correct and complete, that you're using the right algorithm for the job, and that you've written the cleanest, most straightforward, most structured code you possibly can.

If your program meets these criteria, its size and performance will be fine most of the time, without any further attention. Merely by using assembly language you're already getting a two- or threefold speed and size advantage over the fellow using a high-level language. And many of the characteristics that make a program easy to read and maintain will also tend to make it fast—such as the avoidance of spaghetti code with many unnecessary jumps and calls (these have a severe penalty on the Intel 80x86 architecture because they dump the instruction prefetch queue) and the use of simple straightforward machine instructions rather than complex instructions.

Nevertheless, the user's perception of your program's performance should be your overriding concern. As Michael Abrash observed, "Any optimization that is perceptible to the user is an optimization worth doing." Conversely, if your program gets the word-of-mouth reputation among users of being pudgy and pokey, it's very likely to remain unappreciated: As an example, consider the fate of *ToolBook*. Consequently, your program should *appear* to be instantly responsive, even when it's occupied with time-consuming calculations or disk operations. It should keep the screen alive with progress indicators such as dials or thermometers, and allow the user to interrupt lengthy operations safely at any time if he changes his mind and decides to do something else.

Many of the characteristics that make a program easy to read and maintain will also tend to make it fast.

If you do have to resort to the code-hacking and cycle-shaving tricks we mentioned earlier, avoid the misdirection of your time and energy. Bear in mind the natural hierarchy of time scales—register/register operations, memory operations, disk operations, and user interactions—each of which is an order of magnitude (or more) slower than its predecessor. For example, don't waste effort trying to save a few machine cycles in a routine if your program's speed is limited by its access to disk files; try to identify ways to reduce disk I/O instead. It's vital to employ profiling tools to identify the hot spots in your program and concentrate your attention in those areas. And after you make each presumptive optimization, time the results carefully and test, test, test.

In his superb book *Writing Efficient Programs* (Prentice Hall, 1982), Jon Bentley relates a horror story from Bell Labs that we should all take to heart:

"Victor Vyssotsky enhanced a FORTRAN compiler in the early 1960's under the design constraint that compilation time could not be noticeably slower. A particular routine was executed rarely (he estimated during design that it would be called in about one percent of compilations, and just once in each of those) but was very slow, so Vyssotsky spent a week squeezing every last cycle out of the routine. The modified compiler was fast enough. After two years of extensive use the compiler reported an internal error during compilation of a program. When Vyssotsky inspected the code he found that the error occurred in the prologue of the 'critical' routine, and that the routine had contained this bug for its entire production life.'

Vyssotsky made three important errors: He failed to profile the program before he optimized it, so he wasted time optimizing a routine that had no impact on performance; he failed to implement the optimized routine correctly; and he failed to test the optimized routine to see if it worked according to specification.

Now it's not my intent to pick on Mr. Vyssotsky; I've made plenty of far worse blunders in my day, though—since I don't work at Bell Labs—these blunders fortunately don't get immortalized in Jon Bentley's books. But Vyssotsky's experience is a good example of how time and energy can be wasted in the holy cause of optimization, and how a failure to methodically cover all the bases of profiling and validation during optimization will usually come home to roost sooner or later.

THE IN-BOX

Please send your questions, comments, and suggestions to me at any of the following electronic mail addresses:
PC MagNet: 72241,52
MCI Mail: rduncan
BIX: rduncan ■